```
        return SearchBST(BST[p].lc,val);
    else
        return SearchBST(BST[p].rc,val);
}
//在二叉排序树中从结点 p 开始,插入值为 val 的结点
void InsertBST(int &p,int val)
{
    if (p == 0)
    {                              //若二叉排序树为空,则生成并返回一个结点的二叉排序树
        p = NewNode(val); // p 是引用,所以其父结点的 lc 或 rc 值会被同时更新
        return;
    }
    if (val == BST[p].val) //待插入值 val 在原树中已存在
        return;
    else if (val < BST[p].val)
        InsertBST(BST[p].lc,val);
    else
        InsertBST(BST[p].rc,val);
}
//在二叉排序树中删除值为 val 的结点
void DeleteBST(int &p,int val)
{
    if (p == 0)
        return;
    if (val == BST[p].val)
    {                                                  //找到值为 val 的结点
        if (BST[p].lc == 0 && BST[p].rc == 0)//值为 val 的结点的左右子树都为
                                                       空,直接删除 p = 0;
        else if (BST[p].lc == 0) // 值为 val 的结点的左子树为空,用右子树代替
            p = BST[p].rc;
        else if (BST[p].rc == 0) // 值为 val 的结点的右子树为空,用左子树代替
            p = BST[p].lc;
        else
        { // 值为 val 的结点有左右两个子树
            //查找左子树的最大值
            int x = BST[p].lc;
            while (BST[x].rc > 0)
                x = BST[x].rc;
            DeleteBST(BST[p].lc,BST[x].val); //在左子树上将 x 删除,使 x 的子
                                                 树可以填充 x 的位置
            //用结点 x 代替结点 p
            BST[x].lc = BST[p].lc;
            BST[x].rc = BST[p].rc;
            p = x;
```

```
                /*
                //上面四句可以替换为下面两句,直接用结点 x 的值替换原 val 值,再将结点 x
                  删除
                BST[p]. val = BST[x]. val;
                DeleteBST(BST[p]. lc,BST[x]. val);
                */
            }
        }
        else if (val < BST[p]. val) //值 val 比当前结点值小,在当前结点左子树继续查找
            DeleteBST(BST[p]. lc,val);
        else //值 val 比当前结点值大,在当前结点右子树继续查找
            DeleteBST(BST[p]. rc,val);
    }
    //二叉排序树中序遍历
    Void InOrderTraveral(int p)
    {
        if (p == 0)
            return;
        InOrderTraveral(BST[p]. lc);
        cout << BST[p]. val;
        InOrderTraveral(BST[p]. rc);
    }
```

参考词条

1. 二叉树的定义与基本性质
2. 二叉树的遍历：前序、中序、后序

延伸阅读

THOMAS H C，CHARLES E L，RONALD L R，et al. 算法导论(原书第 3 版)[M]. 殷建平，徐云，王刚，等译. 北京：机械工业出版社，2013：161-173.

(佟松龄　叶金毅)

1.3.4　简单图

1.3.4.1　图的定义与相关概念

图是由若干个顶点和若干条边构成的数据结构，顶点是实际对象的抽象，边是对象之间关系的抽象。可以将图形式化表示为二元组。

$$G=(V,E)$$

其中，V 是顶点集，表征数据元素；E 是边集，表征数据元素之间的关系。信息学竞赛

中一般使用 n 表示图中结点的数量，使用 m 表示图中边的数量。

图可以分为无向图（undirected graph）、有向图（directed graph）、混合图（mixed graph）。无向图的边集 E 中的每个元素是一个无序二元组 (u,v) 称作无向边（undirected edge），简称边（edge），其中 u 和 v 称为端点（endpoint）。有向图的边集 E 中的每个元素是一个有序二元组 (u,v)，称作有向边（directed edge）或弧（arc），其中 u 称为弧尾，v 称为弧头。混合图中的边集既有无向边也有有向边。

在无向图中，若任意两个顶点之间都存在边，则该无向图称为无向完全图。n 个顶点的无向完全图，一共有 $n(n-1)/2$ 条边。在有向图中，若任意两个顶点 x、y，既存在 x 到 y 的弧，也存在 y 到 x 的弧，则该有向图称为有向完全图。n 个顶点的有向完全图，一共有 $n(n-1)$ 条弧。

在无向图中，若点 u 与点 v 存在边 (u,v)，则顶点 v 和顶点 u 互称为邻接点。在有向图中，若点 u 与点 v 之间存在一条点 u 指向点 v 的一条弧 (u,v)，则称顶点 u 邻接到顶点 v，顶点 v 邻接自顶点 u。

与顶点相关联的边的数目或者弧的数目称为该顶点的度。在无向图中，顶点的度就是其关联的边的数目。在有向图中，由于与顶点关联的弧具有方向性，因此要区分顶点的入度和出度。入度指以该顶点为弧头的弧的数目，而出度指以该顶点为弧尾的弧的数目，入度与出度之和是该顶点的度。

图中的边或者弧有时带有具有某种意义的数值，称为该边或弧的权。权是边或弧某种属性的数值化描述，边或弧上带权的图可以称为带权图。

在图 $G=(V,E)$ 中，由顶点 u 至顶点 v 的序列，称为路径。路径上边或者弧的数目称为路径长度。如果一条路径中不包含重复结点，则该路径为一条简单路径。如果 u 和 v 相等，则称为一个环。

对于图 $G=(V,E)$ 和 $G'=(V',E')$，若 V' 是 V 的子集，即 $V'\subseteq V$，并且 E' 是 E 的子集，即 $E'\subseteq E$，则称 G' 是 G 的子图。

图中的顶点数也称为图的阶。

图论中的握手定理：对于一个有向图，所有顶点的入度之和等于出度之和，并且和边数也相等；对于一个有向图或者无向图，所有顶点的度数之和等于边数的两倍。

参考词条

树的定义与相关概念

延伸阅读

严蔚敏，吴伟民．数据结构[M]．北京：清华大学出版社，2007：156-160.

（佟松龄　叶金毅）

1.3.4.2　图的表示与存储：邻接矩阵

邻接矩阵存储也称为数组存储，用一个二维数组存放图中的全部边或者弧，以数组

中的下标代表对应顶点。邻接矩阵中的矩阵元素 $A[i][j]$ 存放了顶点 i 和顶点 j 之间的关系。若图是无向图，则 $A[i][j]$ 和 $A[j][i]$ 同时存放顶点 i 与顶点 j 之间的边，若图是有向图，则 $A[i][j]$ 存放顶点 i 到顶点 j 的弧。

对于无权图，邻接矩阵元素 $A[i][j]$ 定义为：

$$A[i][j]=\begin{cases}1, & \text{点 } i \text{ 与点 } j \text{ 之间存在边}\\ 0, & \text{点 } i \text{ 与点 } j \text{ 之间不存在边}\end{cases}$$

对于带权图，邻接矩阵元素 $A[i][j]$ 定义为：

$$A[i][j]=\begin{cases}w_{ij}, & \text{点 } i \text{ 与点 } j \text{ 之间存在权值为 } w_{ij} \text{ 的边}\\ 0, & \text{点 } i \text{ 与点 } j \text{ 之间不存在边}\end{cases}$$

邻接矩阵存储具有以下特征：

(1) 无向图的邻接矩阵是对称矩阵，有向图的邻接矩阵是非对称矩阵；

(2) 对于无向图，其邻接矩阵第 i 行(或第 i 列)非零元素的个数正好是第 i 个顶点的度；

(3) 对于有向图，其邻接矩阵第 i 行非零元素的个数正好是第 i 个顶点的出度，而第 j 列非零元素的个数正好是第 j 个顶点的入度。

对于 n 个顶点的图，采用邻接矩阵存储，需要占用 $n\times n$ 个存储单元，其空间复杂度为 $O(n^2)$。

代码示例

建立一个最多 MAXN 个顶点的无权无向图，代码如下。

```
const int MAXN = 1010;
int G[MAXN][MAXN];
void addEdge(int u,int v)
{ // 加入一条边
    G[u][v] = G[v][u] = 1;
}
```

建立一个最多 MAXN 个顶点的有权有向图，代码如下。

```
const int MAXN = 1010;
int G[MAXN][MAXN];
void addEdge(int u,int v,int w)
{ // 加入一条权为 w 的边
    G[u][v] = w;
}
```

参考词条

二维数组与多维数组

延伸阅读

[1] THOMAS H C，CHARLES E L，RONALD L R，et al. 算法导论(原书第 3 版)[M]. 殷建平，徐云，王刚，等译. 北京：机械工业出版社，2013：341-342.

[2] 严蔚敏，吴伟民. 数据结构[M]. 北京：清华大学出版社，2007：161-163.

（佟松龄　叶金毅）

1.3.4.3　图的表示与存储：邻接表

图的邻接表存储是一种顺序存储与链式存储相结合来存储图信息的存储结构。通常以顺序结构存储建立一个顶点表，记录图中各个顶点的信息；以链式存储结构对图中每个顶点建立一个存储边的线性链表，第 i 个线性链表的结点存储邻接自顶点 v_i 的边及其相关信息。

在无向图的邻接表中，以图中邻接自顶点 v_i 的所有边为结点，构成一个带头结点的线性链表，该线性链表的头结点位置存储于顶点表的第 i 个顶点中。

不带权的无向图的邻接表表示，如图 1.7 所示。

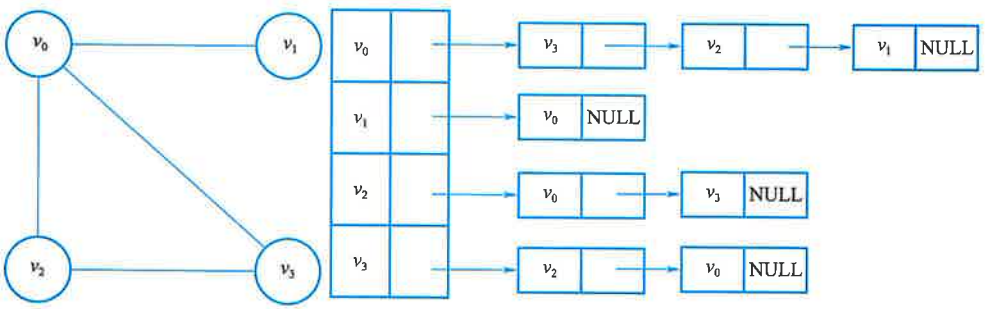

图 1.7　不带权的无向图的邻接表表示

在有向图的邻接表中，以图中顶点 v_i 为弧尾的所有弧作为结点，构成一个带头结点的线性链表，该线性链表的头结点位置存储于顶点表的第 i 个顶点中。

带权的有向图的邻接表表示，如图 1.8 所示。

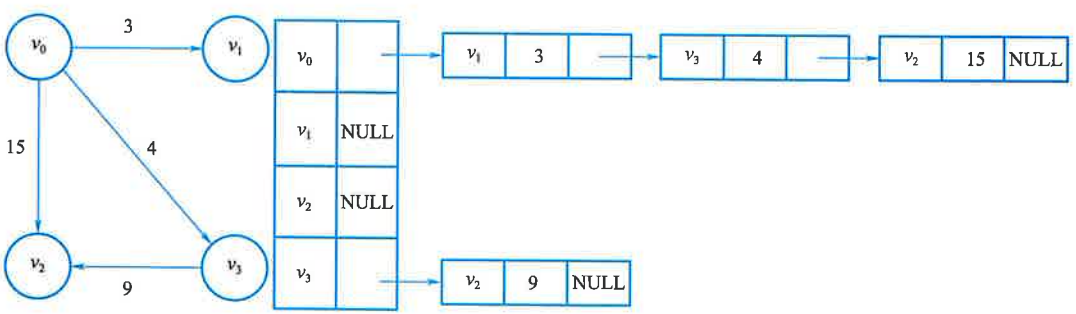

图 1.8　带权的有向图的邻接表表示

顶点表中的每个结点，一般需要定义两个域：